

手写数字识别实验报告

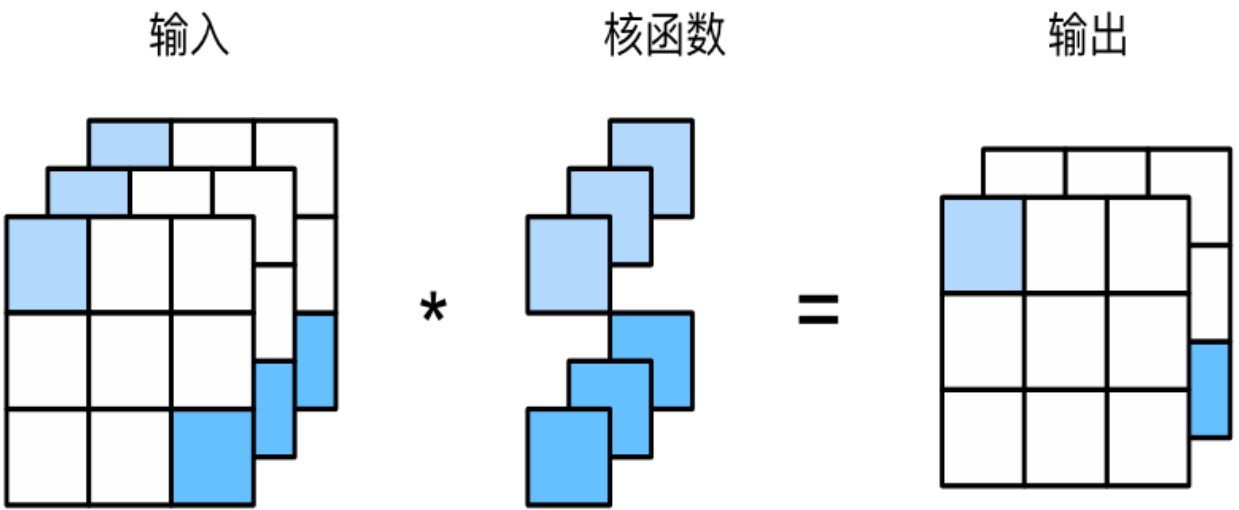
实验目的：

- 1. 掌握卷积神经网络基本原理；
- 2. 掌握 PyTorch（或其他框架）的基本用法以及构建卷积网络的基本操作；
- 3. 了解 PyTorch（或其他框架）在 GPU 上的使用方法。

实验原理：

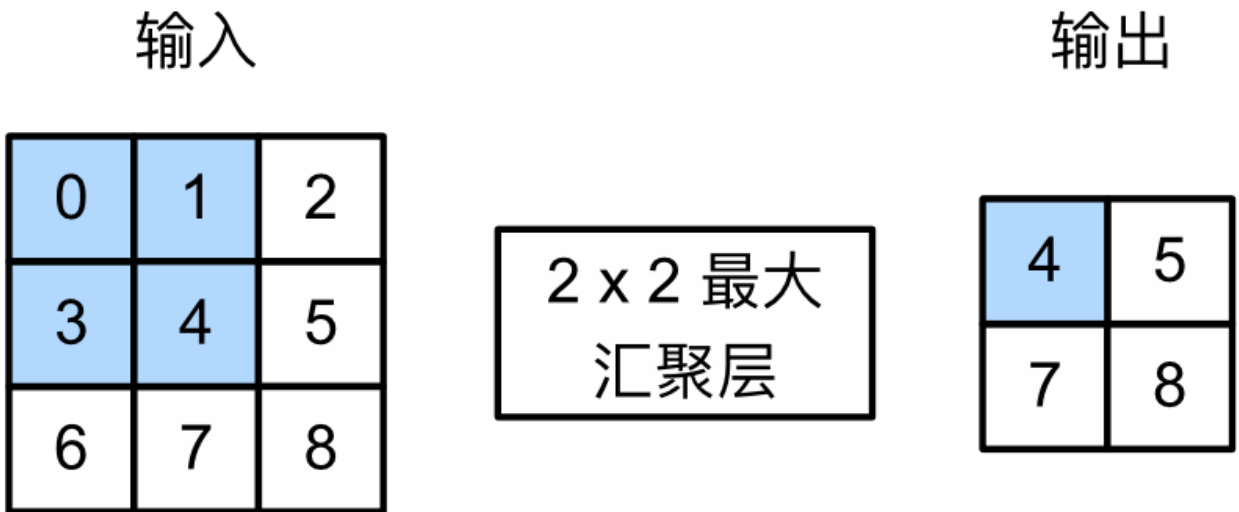
卷积层：

卷积层通过滑动一个固定大小的窗口，称为卷积核或过滤器，对输入数据进行滤波操作，提取出数据中的局部特征。卷积操作可以捕捉输入数据的局部相关性，减少需要训练的参数数量，增强了神经网络对平移、旋转、缩放等操作的鲁棒性。



池化层：

用于降低特征图的空间分辨率，减少需要训练的参数数量，提高网络的鲁棒性和泛化能力。池化操作通常在卷积层之后进行，它可以对卷积层的输出进行空间上的降采样，从而减少输出特征图的大小。



处理流程：通过卷积层进行特征信息的提取，并将提取的信息通过池化层进行下采样操作，减少参数数量，此外通过全连接层来处理输出的维度，使其最终与我们的目的的输出相匹配。

实验步骤：

1. 导入关键的库：

```
import torch
from torch import nn
import torch
import torchvision
import torch.utils.data as Data
from torch import nn
import matplotlib as matplotlib
import matplotlib
import matplotlib.pyplot as plt
```

2. 下载数据集，对数据进行预处理，重写transforms方法。

定义数据集的预处理的方法：

```
transform=transforms.Compose(
    [transforms.ToTensor(),
     transforms.Normalize((0.5),(0.5))
    ]
)
```

```
#下载mnist手写数据集
train_data=torchvision.datasets.MNIST(
    root='./data/',    #用于存放数据，放在当前的位置
    train=True ,       #指定是用于训练的数据，false是指用于测试的数据
    transform=transform, #将数据利用transform方法进行预处理
    download=True
)
test_data=torchvision.datasets.MNIST(
    root='./data/',    #用于存放数据，放在当前的位置
    train=False ,      #指定是用于训练的数据，false是指用于测试的数据
    transform=transform, #将数据利用transform方法进行预处理
    download=True
```

3. 通过torch的Dataloader工具来对数据进行包装，帮助我们打包成按batch划分的数据，方便把我们快速的迭代进行批训练：

```

train_loader=Data.DataLoader(
    dataset=train_data,
    batch_size=50,      #指定训练的batch的大小为50
    shuffle=True        #数据是否都打乱
)
test_loader=Data.DataLoader(
    dataset=test_data,
    batch_size=50,      #指定测试的batch的大小为50
    shuffle=False
)

```

4. 构建我们的模型：

```

class Conv_Net(nn.Module):      #定义一个类该类别继承自我们的nn.module
    def __init__(self):
        super(Conv_Net, self).__init__()
        #声明一个conv1层，该层有容器类构成，其中包含俩层结构，分别是卷积层和池化层
        self.conv1 = nn.Sequential(
            nn.Conv2d(1, 32, 3, 1, 1), #input=【1, 28, 28】    out=[32,28,28]
            nn.ReLU(),
            nn.AvgPool2d(2, 2)         #input=【32, 28, 28】    out=[32,14,14]
        )
        #声明一个conv2层，该层有容器类构成，其中包含俩层结构，分别是卷积层和池化层
        self.conv2 = nn.Sequential(
            nn.Conv2d(32, 64, 3, 1, 1), #input=【32, 14, 14】  out=[64,14,14]
            nn.ReLU(),
            nn.MaxPool2d(2, 2)         #input=【64, 14, 14】    out=[64,7,7]
        )
        self.conv2 = nn.Sequential(
            nn.Conv2d(64, 64, 3, 1, 1), #input=【64, 7, 7】    out=[64, 7, 7]
            nn.ReLU(),
            nn.MaxPool2d(2, 2)         #input=【64, 7,7】       out=[64,3,3]
        )
        #声明一个fc层，该层有容器类构成，其中包含俩线性层构成
        self.fc = nn.Sequential(
            nn.Linear(576, 128),#input=【64*9】    out=[128]
            nn.ReLU(),
            nn.Linear(128, 64),    #input=【128】    out=[64]
            nn.ReLU()
        )
        #定义一个out输出层：
        self.out = nn.Linear(64, 10) #input=【64】    out=[10]
        #定义前向传播的函数
    def forward(self, x):
        x = self.conv1(x)
        x = self.conv2(x)
        x = self.conv3(x)
        x = x.view(x.size(0), -1)
        x = self.fc(x)
        output = self.out(x)

```

```
return output
```

5. 定义优化器和损失函数

```
#创建优化器
# 获取优化器和损失函数
Conv_Net1 = Conv_Net()
optimizer = torch.optim.Adam(Conv_Net1.parameters(), lr=3e-4)#采用adam优化器，学习
率为3e-4
loss_func = nn.CrossEntropyLoss() #定义损失函数为交叉熵损失函数
log_step_interval = 100 # 记录的步数间隔
print(Conv_Net1)
```

6. 定义训练的函数:

```
def train(epoch):
    running_loss = 0.0 # 这整个epoch的loss清零
    running_total = 0
    running_correct = 0
    for step, data in enumerate(train_loader, 0):
        inputs, target = data #将图片和标签从data 中取出
        optimizer.zero_grad() #优化器的梯度清零
        # forward + backward + update
        outputs = Conv_Net1(inputs) #将图片输入网络进行前向传播
        loss = loss_func(outputs, target) #将前向传播的输出和原始的标签求其损失
        loss.backward() #损失反向传播
        optimizer.step() #优化器反向传播梯度并迭代
        # 把运行中的loss累加起来，为了下面300次一除
        running_loss += loss.item()
        # 把运行中的准确率acc算出来
        _, predicted = torch.max(outputs.data, dim=1) #从这个输出的结果的第一个维度开
        始进行torch.max的操作，返回俩个数值，其中第一个为最大概率，第二个为最大概率所对应的下标
        running_total += inputs.shape[0]
        running_correct += (predicted == target).sum().item() #将其和实际的结果对比并
        进行正确的求和，来计算我们的准确率
    if step % 50 == 0:
        correct = 0
        total = 0
        with torch.no_grad(): # 测试集不用算梯度
            for data in test_loader:
                images, labels = data
                outputs = Conv_Net1(images)
                _, predicted = torch.max(outputs.data, dim=1) # dim = 1 列是第0个维
                度，行是第1个维度，沿着行(第1个维度)去找1.最大值和2.最大值的下标
                total += labels.size(0) # 张量之间的比较运算
                correct += (predicted == labels).sum().item()
            acc = correct / total
```

```

        print('%d / %d]: Accuracy on test set: %.1f %% ' % (epoch+1, EPOCH, 100
* acc)) # 求测试的准确率, 正确数/总数

EPOCH = 10
for epoch in range(EPOCH):
    train(epoch)

```

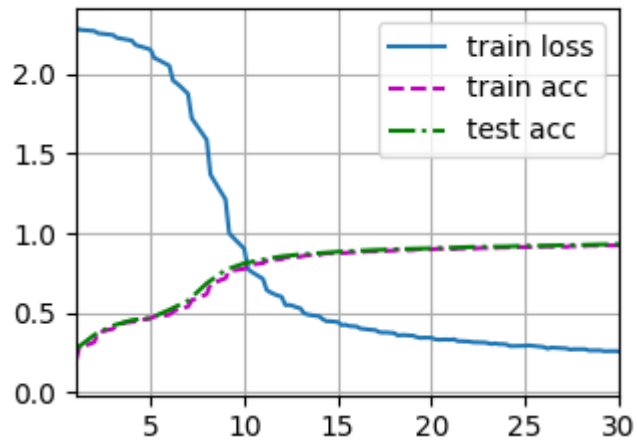
实验的结果:

```

[9 / 10]: Accuracy on test set: 99.1 %
[9 / 10]: Accuracy on test set: 99.0 %
[9 / 10]: Accuracy on test set: 99.0 %
[9 / 10]: Accuracy on test set: 98.9 %
[10 / 10]: Accuracy on test set: 99.0 %
[10 / 10]: Accuracy on test set: 99.0 %
[10 / 10]: Accuracy on test set: 99.1 %
[10 / 10]: Accuracy on test set: 99.0 %
[10 / 10]: Accuracy on test set: 98.8 %
[10 / 10]: Accuracy on test set: 98.8 %
[10 / 10]: Accuracy on test set: 99.0 %
[10 / 10]: Accuracy on test set: 98.8 %
[10 / 10]: Accuracy on test set: 99.0 %
[10 / 10]: Accuracy on test set: 98.9 %
[10 / 10]: Accuracy on test set: 98.9 %
[10 / 10]: Accuracy on test set: 98.7 %
[10 / 10]: Accuracy on test set: 99.0 %
[10 / 10]: Accuracy on test set: 98.8 %
[10 / 10]: Accuracy on test set: 98.9 %
[10 / 10]: Accuracy on test set: 99.1 %
[10 / 10]: Accuracy on test set: 99.1 %
[10 / 10]: Accuracy on test set: 99.1 %
[10 / 10]: Accuracy on test set: 99.0 %
[10 / 10]: Accuracy on test set: 99.0 %
[10 / 10]: Accuracy on test set: 98.9 %
[10 / 10]: Accuracy on test set: 99.0 %
[10 / 10]: Accuracy on test set: 99.0 %
[10 / 10]: Accuracy on test set: 99.1 %

```

运行结果:



实验结论：

通过以上的操作我们的测试的数据集准确率在百分之九十九，达到了我们预期的要求。通过对网络的搭建让我学习到了怎么在PYTORCH框架下进行数据的导入和模型的训练。